LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-410635

A Foundation for the Accurate Prediction of the Soft Error Vulnerability of Scientific Applications

G. Bronevetsky, B. de Supinski, M. Schulz

February 18, 2009

IEEE Workshop on Silicon Errors in Logic - System Effects
Stanford, CA, United States
March 24, 2009 through March 25, 2009

Disclaimer

A Foundation for the Accurate Prediction of the Soft Error Vulnerability of Scientific Applications

Greg Bronevetsky, Bronis R. de Supinski and Martin Schulz
{bronevetsky, bronis, schulzm}@llnl.gov
Center for Applied Scientific Computing,
Lawrence Livermore National Laboratory,
Livermore, CA 94551, USA

Abstract

Understanding the soft error vulnerability of supercomputer applications is critical as these systems are using ever larger numbers of devices that have decreasing feature sizes and, thus, increasing frequency of soft errors. As many large scale parallel scientific applications use BLAS and LAPACK linear algebra routines, the soft error vulnerability of these methods constitutes a large fraction of the applications' overall vulnerability. This paper analyzes the vulnerability of these routines to soft errors by characterizing how their outputs are affected by injected errors and by evaluating several techniques for predicting how errors propagate from the input to the output of each routine. The resulting error profiles can be used used to understand the fault vulnerability of full applications that use these routines.

1 Introduction

In recent years, supercomputing systems have grown dramatically. Systems like BlueGene/L and RoadRunner feature more than 100,000 processors and tens of TBs of RAM and future designs promise to exceed these limits by large margins. Large supercomputers are made from high-quality components, but increasing component counts make them vulnerable to faults, including hardware breakdowns [14] and soft errors [10].

Modern electronics are increasingly susceptible to soft errors [1], with SRAM soft error rates (SERs) growing exponentially as larger and larger memory chips come into use. Current systems typically experience 1,000-10,000 failures per billion hours of operation (FIT) per Mb of memory [3]. A cluster with 1000 processors, each supporting a 10Mb cache with 1600 FIT averages 10 errors per month [3]. Further, soft errors in microprocessor logic are expected to rise in importance [11] in part because the design of flip-flops and latches is similar to that of SRAM cells. Major systems already suffer from the results of soft errors, with ASCI Q experiencing 26.1 CPU failures per week [10] while an L1 cache soft error occurs about once every four hours on the 104K node BlueGene/L system at Lawrence Livermore National Laboratory.

The soft error problem makes it difficult to trust the results of applications that run on large systems. To overcome this difficulty, we must understand how such those errors affect application output, particularly if the errors are not easily detected or corrected. Existing work evaluates application vulnerability by injecting errors and examining their effect on application output. While these techniques can be effective, they can also be very expensive, requiring hundreds or thousands of application runs to achieve a representative sample size.

We explore an alternative to full application fault injection based on the fault vulnerability of the application's individual routines. Specifically, we examine how errors affect the output of individual routines how errors propagate through the routines. This approach supports analysis of the fault vulnerability of applications that use these routines without the cost of a full fault injection experiment. This paper represents an early step in this "modular" error vulnerability analysis by evaluating the fault vulnerability of the major routines from the BLAS [7] and LAPACK [2] linear algebra libraries. We evaluate the errors patterns that result from injecting errors both the original and fault tolerant versions of these routines. We also explore predictions of the error propagation properties of each routine based on machine learning techniques. Our results identify algorithms and training methods that result in accurate predictions for those routines.

[0]This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. (UCRL-ABS-XXXXXX).

2 State of the Art

"Fault injection" is the most common technique to investigate the effect of soft errors: random faults are injected into application state at random points during its execution. These injections can be performed at multiple levels of abstraction, ranging from memory bit-flips [6] to architectural-level simulation [12] to physical experiments [9].

Many researchers have applied fault injection to a few representative systems and applications in order to examine their fault vulnerability and provide a basic understanding of the soft error problem. However, developers need to understand the fault vulnerability of their specific application and it is difficult to extend fault injection results to other applications. Furthermore, fault injection techniques are still very expensive: a single fault injection campaign requires thousands of injections for each variant of the application and input set and a full software-level fault injection study may take tens to hundreds of thousands of CPU hours (Lu and Reed [6] used ∼138,000 CPU-hours to study three parallel applications with reduced input sets), with lower-level studies either much more expensive or much more limited in their scope.

Currently application developers have few options for evaluating the fault properties of their applications. Existing techniques limit them to a few small examples or to extrapolations from full studies of (hopefully) representative applications. However, the increasing impact of soft errors in high performance computing makes a full and detailed understanding of application fault vulnerability increasingly critical. Without this understanding, application users may rely on corrupted application results or waste significant compute cycles due to overly conservative fault tolerance solutions. Thus, we must develop new, efficient techniques to characterize the fault vulnerability of specific applications.

3 Experimental Details

BLAS and LAPACK are popular libraries that contain many commonly used linear algebra operations. In addition to being used directly by many scientific application developers, these libraries are common building blocks of larger libraries such as ScaLAPACK [4] and BLACS [8]. This makes them excellent representatives for the types of libraries used in scientific computing. Although BLAS and LAPACK consist of many routines, we focused on several high-level routines that are likely to be used directly by application developers and contain a non-trivial amount of computational work [1]. In this study we focused on the versions that operate on generic real matrixes of double precision floating point numbers. The names of these routines start with "`DGE`" to signify this configuration (`D` for double and `GE` for general matrix), followed by the name of the routine.

[1]We purposely avoided simple routines such as matrix copy or multiplication by a scalar.

In addition to evaluating the basic operations, we also evaluated fault tolerant versions of these operations that use operation semantics to validate the output of the operation and repeat the operation if an error is detected. With each operation we first run the real operation then run some tester code that validates its correct output. For example, SVD factorization is verified by multiplying the factored matrixes together and comparing the result to the original matrix. If there is a discrepancy, the operation is executed again and if the identical discrepancy appears again we assume that the detection was a false positive due to roundoff error. If the discrepancy is different, we run the operation yet again and return the majority result.

The routines examined in this study are: `DGEMV`, `DGEMM`, `DGER`, `DGESV`, `DGELS`, `DGESVD`, `DGGSVD`, `DGESDD`, `DGEEV`, `DGEGEV`, `DGEES`, `DGGES`, which includes matrix-matrix and matrix-vector multiplication, rank-1 update, linear least squares, SVD factorization and eigenvector routines.

This study was performed on two different platform using the BLAS and LAPACK implementation available inside Intel's Math Kernel Library (MKL). The first platform, referred to in the following as Itanium2/MKL8, consists of 4-way single-core 1.4Ghz Itanium2 nodes with 8GB RAM running MKL version 8. The second platform, referred to as Opteron/MKL10, consists of 4-way dual-core 2.4Ghz Opteron nodes with 16GB RAM running MKL version 10. Both platforms run Chaos Linux, a high-performance Linux variant based on RedHat Enterprise Linux.

We injected all faults using the Sting fault injector based on the Dyninst dynamic instrumentation library [5]. It stops the application at a random points and then flips a single bit in the application's heap, globals, stack or in a register. In this study we focused on heap injection as the error model since all BLAS and LAPACK data is stored on the heap and RAM corresponds to the physically largest soft error target on modern systems.

4 Fault Injection into Library Routines

4.1 Representation of Corruptions

Error patterns in matrixes and vectors are represented as follows. For each entry x_{out} in the error-free output matrix or vector, and entry x'_{out} of the corrupted output matrix or vector, we compute the corruption c as $c = x'_{out}/x_{out}$. We represent value corruptions as multiplicative factors because they closely correspond to the notion of error magnitude that is most relevant to linear algebra calculations. Other metrics such as the number of flipped bits would not be nearly as relevant.

To simplify our analysis we represent matrix/vector corruption patterns as histograms, where the x-axis corresponds to the multiplicative error factors c and the y-axis corresponds to the number of times the given factor occurs in a given corrupted matrix/vector. To bound the number corruption factors used in the analysis, we group similar factors and maintain counts for each group. Figure 1 illustrates this by showing the output error histogram for the `DGEMM` operation with the corresponding multiplicative factors groups below. 0, 1, -1, ∞ and $-\infty$ are assigned their own groups. Factors around 1 and -1 are are assigned to the smaller groups, with $[1-2^{-50}, 1)$ and $(1, 1+2^{-50}]$ being the smallest. Groups double in size as they move away from 1 and -1 until they reach 2, 0 and/or -2. Groups outside [-2, 2] are much larger, increasing at a double exponential rate as they move further from 0. The top number line shows how these groups given equal space and the bottom shows how these groups would map to the real number line.

This grouping scheme directly correlates to bit flips in numbers stored in IEEE floating point number notation. Bucket $\in [0, 2]$ corresponds to flips in the mantissa and buckets $\in [2, 10^{308}]$ correspond to $0 \rightarrow 1$ bit flips in the exponent. $1 \rightarrow 0$ bit flips in the exponent fall into buckets $\in [0, 1]$ and flips in the sign bit falls into the -1 bucket. Other negative buckets correspond to a sign bit flip followed by a flips in other bits. Another property of this representation is that it provides the finest granularity of representation around 1, the multiplicative factor that corresponds to no error. Most application developers have a target error tolerance that naturally occurs due to round-off error, meaning that errors within this tolerance are irrelevant to their results. Thus, fine granularity around 1 gives developers a clear way to identify the probability and distribution of important errors.

4.2 Results of Injection

For our BLAS and LAPACK fault injection experiments we ran each routine 2000 times on a combination of random input matrixes and vectors. We then compared the outputs of each routine to the corresponding error-free outputs and computed the resulting corruption pattern. As our input set we generated 100 random square non-symmetric matrixes using the `DLATMR` LAPACK routine, with the individual numbers sampled from the uniform distribution (`Uni`) in the range [-1, 1]. We used matrixes sizes of 62x62, 125x125, 250x250 and 500x500 and vectors of corresponding sizes ranging from 62x1 to 500x1. Figure 2 shows sample output corruption patterns from three routines from Itanium2/MKL8 (similar results on Opteron/MKL10), from runs on 62x62 inputs on operations `DGGES`, `DGEMM` and `DGESV`.

The four error patterns visible in this figure are representative of the patterns seen in other routines. The output `beta` of `DGGES` only shows positive errors, with a distribution focused around 1 and falling off for very small and large error factors. The spike at 1 is a common feature of the output error histograms and corresponds to a multiplicative error c that corresponds to no error. In `MM`'s output `C` we observe a related pattern with negative multiplicative errors showing either high exponents or only a sign bit flip ($c = -1$). `DGGES`'s output `vsr` shows a pattern that has the full spectrum of errors, with spikes at 1 and -1 and some concentrations towards the extremely small and extremely large multiplicative errors. Negative errors are almost two orders of magnitude less common than positive errors. `DGESV`, on the other hand, shows a significantly different pattern: all the errors are positive and occur not as a raised plateau but in a flat line, with different multiplicative factors being equally likely. `DGESV`'s output `L` shows yet another pattern, where only a few specific error factors are represented.

The major feature of all these graphs is that individual bit-flip errors tend to diffuse themselves into many entries in each routine's outputs. These errors may stay exclusively positive or may take up both signs. Small errors are typically more common than large errors.

Figure 3 shows changes in the output error histograms of `DGESVD` (SVD factorization) outputs `beta` and `V` as the input sizes grow from 62x62 to 500x500. These outputs, which are representative of the outputs of other operations, show how little error patterns vary across input sizes. `beta` shows practically no change, while `V`'s pattern evolves very gradually, reaching steady state by 250x250. This data highlights that there exists a weak dependence between the input size and the output error pattern that drops off for larger matrixes. This indicates that although input size matters, studying the error behavior of BLAS and LAPACK routines at small inputs will result in small error. Based on this observation, this study focuses on 62x62 inputs, since they are computationally more tractable.

From the perspective of the application developer the most important thing about an operation's output error pattern is not its exact shape but rather the probability of the output having an error that is large enough to compromise the integrity of the application's results. Because different applications have different levels of "acceptable" error due to natural round-off errors and errors in the input, different applications have different tolerances for error due to random faults. Figure 4(a) shows the probability for each operation that a fault injection will result in an error that is larger than a given tolerance, for tolerances ranging from multiplicative errors of $2e-14$ to .75. The output error rates vary dramatically between operations, with the eigenvector operations having $> 60\%$ chance of outputting an error at tolerance $4e-13$, while `DGEMM`, `DGER` and `DGELS` have only a 2.5%,

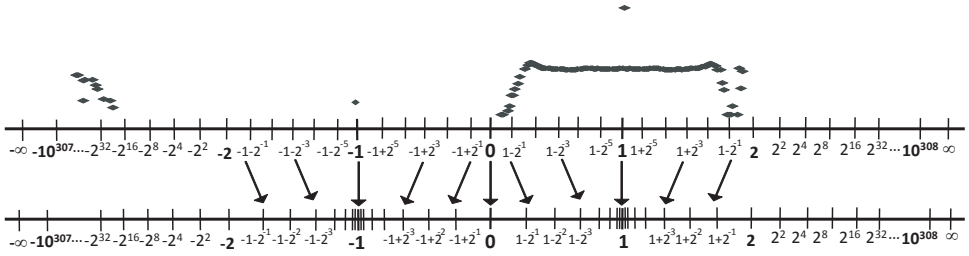

Figure 1: Multiplicative factor groups

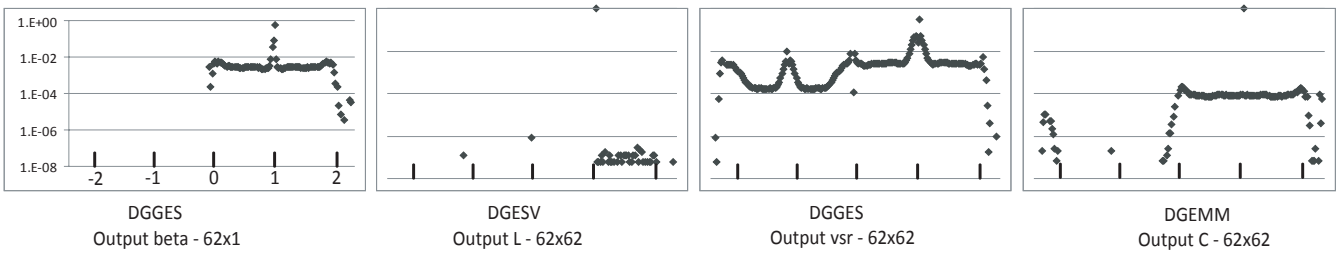

Figure 2: Sample error histograms

.5% and .012% chance of outputting an error at the same tolerance. As the application's tolerance for errors rises, the error rates drop steadily. However, while most operations drop to below 10% error at the very loose tolerance of .75, the eigenvector operations only reach 32% chance of error for `DGEES` and `DGEEV` and 14% chance for `DGGEV` and `DGGES`. For most tolerances the error rates of most operations are $> 10\%$.

Figure 4(b) shows the error rates of the fault tolerant variants of each operation as a fraction of the error rate of the original operation for the same range of tolerances. The fault tolerant operations reduce the error rates dramatically by a factor of two for small tolerances and more than ten for large tolerances. However, they do not erase errors completely because the testers signal an error only if they detect a discrepancy of at least $1e-7$ between two values that should be equal. The detection bound was set to this value because it is the tightest bound where the false positive rate (probability of ordinary round-off resulting in a failed test) rose above 5%.

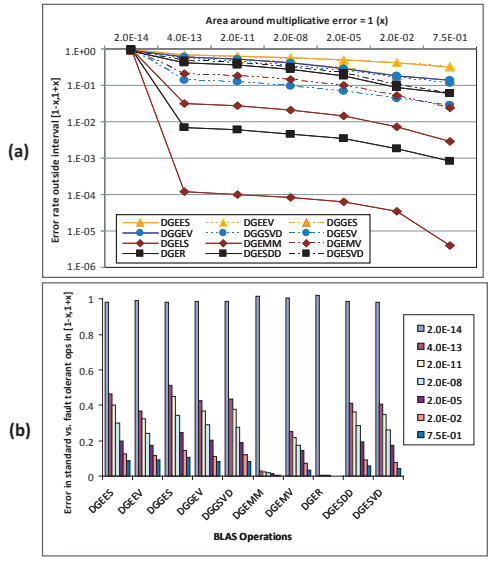

Figure 4: Error rates of regular and fault tolerant operations for different tolerances

These results show that a single injected fault can have a dramatic effect on the output of BLAS and LAPACK operations, with a single 1-bit flip resulting in the output being corrupted in many places, the corruptions ranging widely in magnitude. This means that soft errors are a significant problem for numeric routines and can affect the integrity of application results.

Note that the output error patterns were identical on our two platforms: IA64/MKL8 and Opteron/MKL10. This suggests that the error vulnerability of MKL BLAS and LAPACK has little dependence on the underlying architecture. Since the algorithms used by the two versions of MLK are the same, it means that algorithmic details have the largest influence on error propagation while low-level details such as exact choice and order of instructions is much less important.

5 Fault Propagation

5.1 Training Propagation Predictors

The results discussed above help us understand the effect of soft errors on individual routines. The next step is use this information to understand how these errors are propagated through the application. The full effect of this propagation completely depends on how the affected state is used by the application. Thus, a given error may have no effect because it affects state that will never be read again or it may have a dramatic effect if it is amplified by routines that use it in their input. Since it is very difficult to analytically determine the error propagation properties of complex routines, the only practical alternative is to empirically compute how each routine's propagates error training a predictor function from sample input and output error patterns. Such predictors can then be used to analyze the error vulnerability properties of full applications. Producing accurate predictions of operation output error histograms in a wide variety of applications requires the right choice predictor algorithm and the right choice of sample input error patterns on which to train it. In this study we evaluated three different predictor algorithms on five different training set classes.

The predictor algorithms employed were:

- `LinSq`: linear least squares,
- `SVM`: support vector machines, using the linear, 2^{nd} degree polynomial, sigmoid and rbf kernel functions (for rbf, gamma = 0.1, 1.0 and 10.0), and
- `ANN`: artificial neural nets, using the linear, gaussian, gaussian symmetric and sigmoid transfer function with either 3, 10 or 100 hidden layers.

We trained each algorithm on a list of input/output error histograms, where the input error histograms were drawn from one of five distributions. In each run we corrupted a single entry of every operation input from the given distribution. The distributions were:

- `DataInj`: Multiplicative error that corresponds to a flip in a single bit of an IEEE double floating point number.
- `DataInj-R`: Recursive version of `DataInj` where we sample from the distribution of output errors that result from using `DataInj` to corrupt the input.

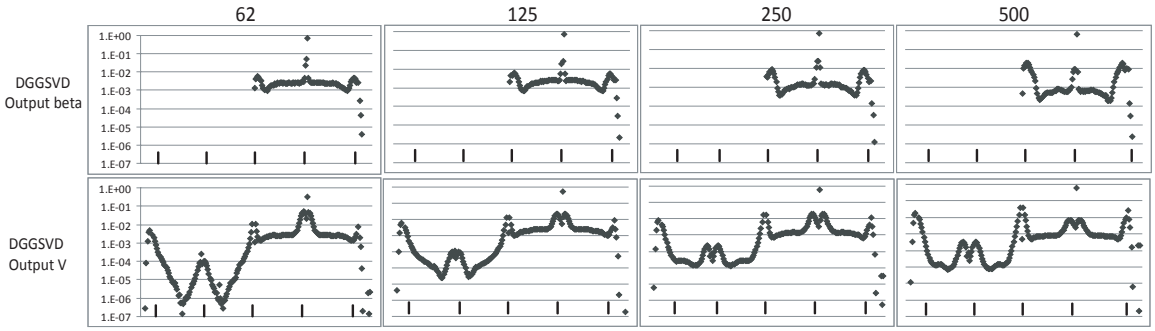

Figure 3: Error histograms scaling with input size

- `DataUni`: Multiplicative error chosen uniformly at random from the range [-100, 100].
- `DataUni-R`: Recursive version of `DataUni`.
- `Inj-R`: Recursive version of error injection where we sample from output error histograms that result from fault injection; when drawing errors from this distribution we first randomly pick an operation and then sample from it output error histogram.

Each routine was run on ∼2000 random input corruptions.

5.2 Predictor Accuracy

Figure 5 provides a higher-level picture of the accuracy of the different predictor options and training sets. Figure 5 shows the accuracy of different predictors, which is measured as the ratio of the Earth Mover Distance [13] (EMD) between predicted and real output error patterns and the EMD of the real error and the error pattern of a single 1-bit error (the trivial error hypothesis); lower is better. Figure 5(a) shows the predictor accuracy relative to their respective training sets and averaging over all the operations. It becomes clear that neural nets perform consistently poorly regardless of the training set. `LinSq` performs best when trained on the `DataInj` distribution while `SVM` is best when trained on the `DataInj` and `DataInj-R` distributions, which behave similarly. `SVM` with the linear and polynomial kernels reach the best fits, although the `LinSq` predictor performs more consistently. Figure 5(b) shows the same data but plots the predictors against the operations, averaging over the training sets. Again `ANN` shows consistently poor performance. `LinSq`, `SVM`-linear and `SVM`-polynomial show similar performance, with little consistency across different operations. `SVM`-rbf shows intermediate performance between the two groups.

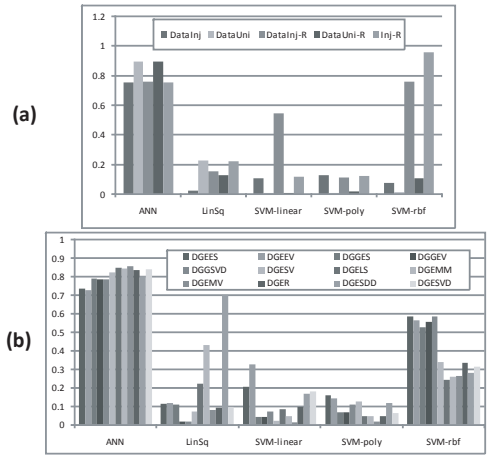

Figure 5: Accuracy of all configurations, relative to error of `DataInj` distribution

6 Summary

This paper has presented an experimental evaluation of error vulnerability of common scientific computing libraries, BLAS and LAPACK. By looking at application error vulnerability in a modular fashion this work supports future efforts to analyze the error vulnerability of full applications by aggregating these per-routine profiles into full application profiles. We evaluated the fault vulnerability of both the regular as well as fault tolerant versions of 12 major BLAS/LAPACK routines, showing that checking routine output significantly improves their reliability and that the error patterns of various routines change little with larger input sizes. Further, we evaluated the possibility of predicting the fault propagation properties of scientific library routines by looking at three different machine learning algorithms that were trained on five different input error patterns. In the process we showed that the best accuracy was exhibited by linear least squares and support vector machines with two kernels and that the choice of input error patterns used to train the predictor was important.

References

[1] International technology roadmap for semiconductors. White paper, ITRS, 2005.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[3] R. C. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and Materials Reliability*, 5(3):305–316, September 2005.

[4] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[5] Bryan R. Buck and Jeffrey K. Hollingsworth. An api for runtime code patching. *Journal of High Performance Computing Applications*, 14(4), 2000.

[6] Charng da Lu and Daniel A Reed. Assessing fault sensitivity in mpi applications. In *Supercomputing*, November 2004.

[7] J. Dongarra. Basic linear algebra subprograms technical forum standard. *International Journal of High Performance Applications and Supercomputing*, 16(1):1–111, June 2002.

[8] Jack J. Dongarra and R. Clint Whaley. A user's guide to the blacs. Technical report, 1993.

[9] Austin Lesea and Peter Alfke. Rosetta: Soft-error test update. White paper, Xilinx Inc., 2004.

[10] Sarah Michalak, Kevin W. Harris, Nicolas W. Hengartner, Bruce E. Takala, and Stephen A. Wender. Predicting the number of fatal soft errors in los alamos national laboratorys asc q supercomputer. *IEEE Transactions on Device and Materials Reliability*, 5(3), 2005.

[11] S. Mitra, Ming Zhang, T.M. Mak, N. Seifert, V. Zia, and Kee Sup Kim. Logic soft errors: a major barrier to robust platform design. In *IEEE International Test Conference*, page 10, 2005.

[12] Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steve Reinhardt, and Todd Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *MICRO*, December 2003.

[13] Yossi Rubner, Carlo Tomasi, and Leonidas J. Guibas. A metric for distributions with applications to image databases. In *International Conference on Computer Vision*, pages 59–66, 1998.

[14] Bianca Schroeder and Garth A. Gibson. A large-scale study of failures in high-performance computing systems. In *In Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2006.